Semaphores
Basic Idea

• An object with an integer value
• Manipulate with two POSIX routines: sem_wait() and sem_post()
• Initial value of the semaphore determines its behavior.

• sem_init()
  • 1: The initial value of semaphore
  • 0: Semaphore is shared among all threads of a process

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```
Basic Idea

- `sem_wait()`
  - A thread waits in a queue when semaphore is negative.
  - The negative value indicates the number of waiting threads.

```c
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}
```

```c
int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```
Binary Semaphores (Locks)

- Use semaphore as a lock
- What is the value of X?

```c
sem_t m;
sem_init(&m, 0, X);
sem_wait(&m);
//critical section here
sem_post(&m);
```
Binary Semaphores (Locks)

- Lock has two states: held and not held.
- Semaphore that functions like a lock is called binary semaphore.

<table>
<thead>
<tr>
<th>Value of Semaphore</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>call sema_wait()</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>sem_wait() returns</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>(crit sect)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>call sem_post()</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>sem_post() returns</td>
<td></td>
</tr>
</tbody>
</table>
# Binary Semaphores (Locks)

<table>
<thead>
<tr>
<th>Value</th>
<th>Thread 0</th>
<th>State</th>
<th>Thread 1</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
<td>Running</td>
<td>Ready</td>
<td>Ready</td>
</tr>
<tr>
<td>1</td>
<td>call sem_wait()</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>sem_wait() retrans</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>(crit set: begin)</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Interrupt; Switch → T1</td>
<td>Ready</td>
<td></td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td>Ready</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>call sem_wait()</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>decrement sem</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>(sem &lt; 0) → sleep</td>
<td>sleeping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>Switch → T0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>(crit sect: end)</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>call sem_post()</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>increment sem</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>wake(T1)</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>sem_post() returns</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Interrupt; Switch → T1</td>
<td>Ready</td>
<td></td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td>Ready</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>(crit sect)</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>call sem_post()</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>sem_post() returns</td>
<td>Running</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Semaphores for Ordering

Order events in a concurrent program

- A thread waits for an event to happen.
- Another thread makes that event happen and signals (wakes up) the waiting thread.

```c
void * child(void *arg) {
    printf("child\n");
    sem_post(&s);  // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X);  // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s);  // wait here for child
    printf("parent: end\n");
    return 0;
}
```
### Parent Waiting for Child (Case 1)

- Parent calls `pthread_create()` but child does not run immediately.
- Parent calls `sem_wait()` before child calls `sem_post()`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Parent</th>
<th>State</th>
<th>Child</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create(Child)</td>
<td>Running</td>
<td><em>(Child exists; is runnable)</em></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>call sem_wait()</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>-1</td>
<td>decrement sem</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>-1</td>
<td>(sem &lt; 0)→sleep</td>
<td>sleeping</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>-1</td>
<td>Interrupt; Switch→Child</td>
<td>sleeping</td>
<td>child runs</td>
<td>Running</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>sleeping</td>
<td>call sem_post()</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>sleeping</td>
<td>increment sem</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Ready</td>
<td>wake(Parent)</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Ready</td>
<td>sem_post() returns</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Ready</td>
<td>Interrupt; Switch→Parent</td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Running</td>
<td>sem_wait() returns</td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
</tbody>
</table>
## Parent Waiting for Child (Case 2)

- Child runs right after parent calls `pthread_create()`.
- Child calls `sem_post()` before parent calls `sem_wait()`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Parent</th>
<th>State</th>
<th>Child</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create(Child)</td>
<td>Running</td>
<td><em>(Child exists; is runnable)</em></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td><em>Interrupt; switch→Child</em></td>
<td>Ready</td>
<td>child runs</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td><em>Interrupt; switch→Child</em></td>
<td>Ready</td>
<td>call <code>sem_post()</code></td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td><em>Interrupt; switch→Child</em></td>
<td>Ready</td>
<td>increment sem</td>
<td>Running</td>
</tr>
<tr>
<td>0</td>
<td><em>Interrupt; switch→Child</em></td>
<td>Ready</td>
<td><code>wake(nobody)</code></td>
<td>Running</td>
</tr>
<tr>
<td>1</td>
<td><em>Interrupt; switch→Child</em></td>
<td>Ready</td>
<td><code>sem_post()</code> returns</td>
<td>Running</td>
</tr>
<tr>
<td>1</td>
<td><em>parent runs</em></td>
<td>Running</td>
<td><em>(Child exists; is runnable)</em></td>
<td>Ready</td>
</tr>
<tr>
<td>1</td>
<td>call <code>sem_wait()</code></td>
<td>Running</td>
<td><em>Interrupt; Switch→Parent</em></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td>decrement sem</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td><em>(sem ≥ 0)→awake</em></td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
<tr>
<td>0</td>
<td><code>sem_wait()</code> returns</td>
<td>Running</td>
<td></td>
<td>Ready</td>
</tr>
</tbody>
</table>
## Example Program

<table>
<thead>
<tr>
<th>Main function</th>
<th>Thread function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Thread 0</td>
<td>Print that this thread started.</td>
</tr>
<tr>
<td>Create Thread 1</td>
<td>Signal the main function that this thread is ready</td>
</tr>
<tr>
<td>Create Thread 2</td>
<td>Execute the next instructions</td>
</tr>
<tr>
<td>Wait for threads to get started</td>
<td></td>
</tr>
<tr>
<td>Print that all threads started.</td>
<td></td>
</tr>
</tbody>
</table>
## Example Program

**Global variable**  
sem_t s;

<table>
<thead>
<tr>
<th>Main function</th>
<th>Thread function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_init(&amp;s, 0, 0);</td>
<td>Print that this thread started.</td>
</tr>
<tr>
<td>pthread_create(...);</td>
<td>sem_post(&amp;s);</td>
</tr>
<tr>
<td>pthread_create(...);</td>
<td></td>
</tr>
<tr>
<td>pthread_create(...);</td>
<td>Execute the next instructions</td>
</tr>
<tr>
<td>sem_wait(&amp;s);</td>
<td></td>
</tr>
<tr>
<td>sem_wait(&amp;s);</td>
<td></td>
</tr>
<tr>
<td>sem_wait(&amp;s);</td>
<td></td>
</tr>
<tr>
<td>Print that <em>all threads started</em>.</td>
<td></td>
</tr>
</tbody>
</table>
The Producer/Consumer (Bounded Buffer) Problem

```c
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value;  // line f1
    fill = (fill + 1) % MAX;  // line f2
}

int get() {
    int tmp = buffer[use];  // line g1
    use = (use + 1) % MAX;   // line g2
    return tmp;
}
```
The Producer/Consumer (Bounded Buffer) Problem

```c
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);  // line P1
        put(i);
        sem_post(&full);   // line P2
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);   // line C1
        tmp = get();        // line C2
        sem_post(&empty);   // line C3
        printf("%d\n", tmp);
    }
}
```
The Producer/Consumer (Bounded Buffer) Problem

• **empty** and **full** semaphores indicate
  - Whether buffer is full or empty
  - Producer waits for the buffer to become empty.
  - Consumer waits for the buffer to become full.

• Assume MAX=1

```c
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX);
    sem_init(&full, 0, 0);
    // ...
}
```
The Producer/Consumer (Bounded Buffer) Problem

• Assume MAX > 1
• Two producers: \( P_a \) and \( P_b \)
• \( P_a \) runs first, but \( P_b \) interrupts \( P_a \) overwrites the data written by \( P_a \).

```c
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value;  // line f1
    fill = (fill + 1) % MAX;  // line f2
}

int get() {
    int tmp = buffer[use];  // line g1
    use = (use + 1) % MAX;  // line g2
    return tmp;
}
```
Adding Mutual Exclusion

- Filling buffer and incrementing index in the buffer are part of a critical section.

- Use binary semaphore

- Prevent a producer to ever attempt filling a slot of buffer while another producer is filling buffer.
Adding Mutual Exclusion

sem_t empty;
sem_t full;
sem_t mutex;  // mutex is initialized to one

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);  // line p0 (NEW LINE)
        sem_wait(&empty);  // line p1
        put(i);
        sem_post(&full);  // line p2
        sem_post(&mutex);  // line p3
        sem_post(&mutex);  // line p4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);  // line c0 (NEW LINE)
        sem_wait(&full);  // line c1
        int tmp = get();  // line c2
        sem_post(&empty);  // line c3
        sem_post(&mutex);  // line c4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
Deadlock

• A consumer acquires the mutex lock, but waits for the *full* condition.
  – The consumer holds the lock.

• A producer can satisfy the full condition, but waits for the release of the mutex lock.

• Producer and consumer wait for each other in a cycle.
Solution

sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);                // line p1
        sem_wait(&mutex);                // line p1.5 (MOVED MUTEX HERE...)
        put(i);
        sem_post(&mutex);                // line p2
        sem_post(&full);                 // line p2.5 (... AND HERE)
        sem_post(&full);                 // line p3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);                // line c1
        sem_wait(&mutex);                // line c1.5 (MOVED MUTEX HERE...)
        int tmp = get();                // line c2
        sem_post(&mutex);                // line c2.5 (... AND HERE)
        sem_post(&empty);                // line c3
        printf("%d\n", tmp);
    }
}
Reader-Writer Locks

• Data structure operations may need different locks.
• Concurrent inserts and lookups in linked lists
  – insert: changes (updates) the list.
  – lookup: reads from the list.
    ➢ *insert* requires a lock.
    ➢ Many concurrent lookups
Reader-Writer Locks

typedef struct _rwlock_t {
    sem_t lock;       // binary semaphore (basic lock)
    sem_t writelock;  // used to allow ONE writer or MANY readers
    int readers;      // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
}
Reader-Writer Locks

```c
rw->readers++;  
if (rw->readers == 1)  
    sem_wait(&rw->writelock); // first reader acquires writelock  
sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) { Not fair to writers  
sem_wait(&rw->lock);  
rw->readers--;  
if (rw->readers == 0)  
    sem_post(&rw->writelock); // last reader releases writelock  
sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {  
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {  
    sem_post(&rw->writelock);
}
```
The Dining Philosophers

while (1) {
    think();
    getforks();
    eat();
    putforks();
}

Implement getforks() and putforks()

• Without deadlock (no philosopher starves)
• High concurrency (many philosophers eat)
The Dining Philosophers

void getforks() {
    sem_wait (forks[left(p)]);
    sem_wait (forks[right(p)]);
}

void putforks () {
    sem_post (forks[left(p)]);
    sem_post (forks[right(p)]);
}

• First, acquire the left fork, then the right fork.
• Deadlock occurs.
  ▪ Each philosopher grabs a fork on their left before another philosopher grabs that fork on their right.
  ▪ Each philosopher holds a fork, but is stuck waiting for another.
void getforks() {
    if (p == 4) {
        sem_wait (forks[right(p)]);
        sem_wait (forks[left(p)]);
    } else {
        sem_wait (forks[left(p)]);
        sem_wait (forks[right(p)]);
    }
}

<table>
<thead>
<tr>
<th>Philosopher</th>
<th>Left fork</th>
<th>Right fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>f0</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>f1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>f3</td>
<td>f4</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>f2</td>
<td>f3</td>
</tr>
<tr>
<td>P1</td>
<td>f1</td>
<td>f2</td>
</tr>
<tr>
<td>P0</td>
<td>f0</td>
<td>f1</td>
</tr>
<tr>
<td>P4</td>
<td>f4</td>
<td>f0</td>
</tr>
</tbody>
</table>